Qualitative Image Sorting

Christopher Lindner
University of Wisconsin – Madison
clindner@wisc.edu

Spencer Buyansky
University of Wisconsin – Madison
buyansky@wisc.edu

Abstract

A common way to describe images is with binary attributes; however this method is restrictive. You can say that one image is "urban" and another is "natural" but how do you distinguish which is "more urban" or "more natural" than another? What if the image contains both settings? Inspired by a paper by Devi Parikh and Kristen Grauman, we propose a way to rank images based on more qualitative attributes. This can be used to sort images in a collection. For our implementation we will use the example of "natural" vs. "urban" images, but if implemented properly, it is possible to qualitatively analyze different potential attributes. We will use various quantifiers and our own ranking algorithm to rank images based on what we calculate their setting or subject to be. Our goal is to have the best sort possible, and tweak it to the point where it can sort as well as a human could manually. It may be hard to define what makes an image urban for example, but we will attempt using techniques like line recognition and analyzing image content and general color schemes. We will make use the Flickr API to download a sample set of images to run though our program which is coded in MatLab.

1. Introduction

Work has been done to map low level image features and use these as a basis for defining binary attributes. These attributes can be used to compare images. Binary attributes are not usually what humans would use to compare images because the human visual system is more complex and semantic. We use words to describe images as being things like "manmade" or "open" or "natural." A higher level descriptor for images would provide better comparisons than binary ones. This will allow us to differentiate images that are very similar or are mixture of two image types.

a. Natural

b. ?

c. Urban

Figure 1. This shows the problem with binary attributes and mixed image types. Our idea is to implement a sort with qualitative comparisons that puts the images in relative order.

2. Motivation

There are millions of images uploaded to sites like Flickr, Imgur and Facebook every day. These websites often have ways to tag and describe image content but many times this metadata is not sufficient to find certain images or sort and categorize them. We believe that we can use relative attributes in images to help sort large collections of images to help us better categorize them. This is all done with minimal human interaction, and the results are hopefully similar to how a human would sort the images. We would then have online databases that are easier to navigate and search. Our particular implementation has been designed to sort between natural and urban images. It could be tweaked to compare different kinds of scenes if needed.

3. Problem Statement

Given a collection of images from an online photo sharing site such as Flickr, create an application that will qualitatively sort the collection based on relative attribute comparisons. The application must download a collection using Flickr's API and sort the images similar to how most humans would on a scale of most natural to most urban with mixed images in between. We will try to develop and implement our own algorithm to make the relative comparison.

4. Related Work

Relative Attributes: The idea behind using relative attributes over binary attributes is from a paper by Devi Parikh and Kristen Grauman. Their main contributions were learning visual attributes from a training set and using that to make relative comparisons. They also explained how one might compare images in reference to example images or categories of images. They emphasized how this type of comparison yields much better results than using binary comparisons.

This paper was our primary inspiration for this project. It also gave us the idea to use some sort of learning algorithm, however our implementation is very simplified. The relative attributes paper had many more ranking functions than ours; however, they implemented two different applications, one that compares attributes of faces and another that compares urban and natural images. [1]

Line Detection: One of the key descriptors we focused on in our relative comparison was the number of vertical lines in an image. There are many different ways to detect lines in images. We spent a long time trying to find a method that would find lines in urban images and fewer lines in images that are more natural. We looked through many different existing Hough Transform implementations for detecting and counting lines. [2] We initially tried using this method and some open source code for MatLab. The code we downloaded, Hough Lines, was certainly capable of finding lines but with the resolution and type of images we were using the function did not find very many lines. On a low-res urban image it would locate

around five to ten lines. This was not a good enough result for our application. Some of the urban images had upwards of one hundred lines in actuality. In the end we decided to implement our own algorithm for finding lines; however, the trials we ran with Hough Lines changed our approach.

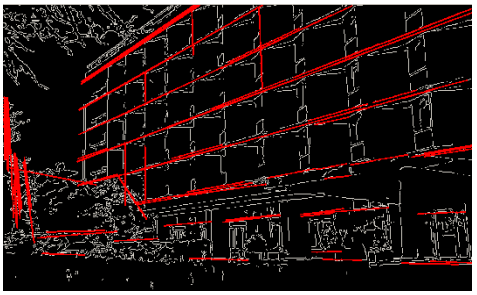


Figure 2. Results from a high-res hough line detection. [3]

Image Descriptors: Much work has been done in finding semantically similar images and even using that to categorize and sort images. The first paper on Relative Attributes used the GIST descriptor along with a color histogram for image features. This could have provided another way to tell if an image is "outdoors" or "open" or resembles one of our two attributes more than the other. We decided not to use GIST so we would implement our own comparator. We have considered adding it as an extension of our implementation in the future.

5. Method

Our method has eight functions, scripts and applications that each play a specific role in the final sorting of the collection of images. They can be split up into three main parts: color scheme detection, vertical line analysis, and the main sorting application. The main application uses the color scheme analysis and line detection to make relative comparisons while sorting the input collection.

5.1 Calculating Natural Color Schemes: We wanted to train our algorithm to know the average green value for natural images to help make relative comparisons. The "green value" is calculated in the *calcColor.m* function which finds the average red, green, and blue values for each color channel in the image and stores these averages in red, green, and blue, respectively. It then returns a value corresponding to how close the average color scheme is to green by using the formula:

$$Green_{Approx} = \overline{green} - \frac{\overline{red} + \overline{blue}}{2} + 1$$

The R, G & B components are in double form (0-1). We add one to the final approximation to avoid the possibility of a negative result so the final range for Green Approx. is (0-2).

5.2 Training for Natural Color Schemes: For color scheme comparison to be effective, a standard value for natural color schemes must be established. The MatLab function *colorTrain.m* accomplishes this by using a set of training images to find an average green value. This value is used in the relative image comparisons. The training images are downloaded from Flickr using a text based boolean search of "`(nature AND green) -portrait`". This search provided the most consistent results, and was chosen due to green being the most consistent color

feature among natural images. At first we believed “nature AND green” would be a sufficient to find suitable training images for the algorithm, however the addition of “-portrait” became necessary because many people tag their portraits as “natural” meaning they are not digitally altered. These photos skewed the color detection training, so they were removed through the boolean search. After downloading and reading in the images, the function does a simple average of the values returned by calcColor.m for each image to find the final average green value of the training images.

Given *n* training images: $avgGreen_{training} = \frac{\sum_0^n calcColor(n)}{n}$

5.3 Finding Vertical Edge Points: As with the presence of a green color scheme, number of vertical lines is another reliable indicator of how natural or urban an image is. The function findEdgePoints finds vertical lines within an image by taking the gradient of the image, $[DX\ DY] = gradient(image)$; similar to finding the energy of an image. This returns a difference matrix, [DX DY], which contains the difference in the X direction and Y direction, respectively, for each pixel. For the energy image these values are usually summed, but we wanted to emphasize vertical lines over horizontal lines. For this reason, we use the formula:

$$Vertical\ Energy = -2 * |DY| + 2 * |DX|$$

This ensures that the X-differential (vertical) is emphasized over the Y-differential (horizontal) for each pixel. For the final computed edge image, a threshold of .3 is set to make it more likely that strong vertical lines are found. This is especially useful in low resolution or highly compressed image as it helps our algorithm not find vertical lines in the artifacted parts of the image. The return value is a ‘2 x n’ array of points that lie on a vertical line that passes the threshold.

5.4 Finding Lines from Edge Points: Finding the vertical edge points does not guarantee the existence of vertical lines. To better ensure that a line exists between several points we needed a function that would find lines based on the potential line points passed to it from findEdgePoints. As stated before, we attempted an implementation of a Hough Transformation, but this proved to be a bad match for our images due to their low resolution. The Hough Transformation would not find enough lines to differentiate most images, especially those which were a mix of both natural and urban scenery. We then turned to linear regression which ended up being our final implementation.

Treating the array of points returned by findEdgepoints as a scatterplot, we randomly select a pair of endpoints then fit as many of the remaining points to the line formed by those edge-points. The process starts with finding the slope and y-intercept of the line using simple algebra:

$$slope = \frac{Y_2 - Y_1}{X_2 - X_1} \qquad Y_{int} = -X_1 * slope + Y_1$$

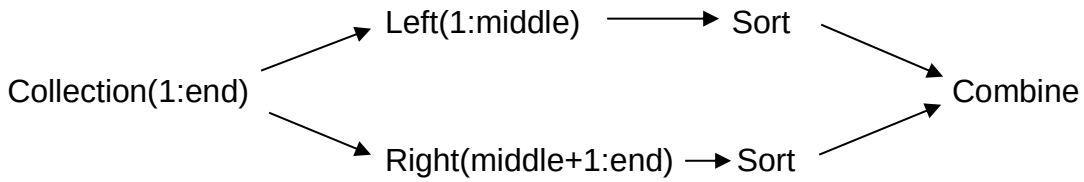
Then for all the remaining edge points we attempt to fit the point to the line formed by the edge-points by plugging in their X value into the equation given by the slope and Y-intercept. If the fit value is within ±1 of the original y value then the point is considered a “fit point.” A line must contain 75 or more “fit points” for it to be counted as a line in our method. This linear regression is performed on 10000 randomly chosen pairs of edgepoints. The function returns the number of lines found in the image and the slope and y intercept of each line.

5.5 Relative Comparison: After calculating the relative color scheme and number of lines within an image, a function is needed to use this data to decide whether or not an image is more urban than another image. We decided that the number of lines was a better indicator of how urban an image is, since even an urban image potentially has a green color scheme (e.g. a green building). A threshold of eight lines is set so that if the difference of number of lines in the two images is within eight, the function then checks to see which image is closer to the trained value for natural green, and returns that as the more natural image. If the difference in number of lines is greater than the threshold, then the image with more lines must have a green value that is more than .1 greater than the other image to override the line count and be considered more natural. A difference of .1 with our measurement of color is very significant and is enough to declare that an image is more natural than another image

Pseudo-code for Comparison Algorithm:

```
if (The Difference in # Vert. Lines) > 8
    if (# Lines in Image 1) > (# Lines in Image 2)
        if (Image 1 Green Value) > (Image 2 Green Value + .1)
            Return Image 1 More Natural
        else
            Return Image 2 More Natural
        end
    else
        if (Image 2 Green Value) > (Imag1 Green Value + .1)
            Return Image 2 More Natural
        else
            Return Image 1 More Natural
        end
    end
else
    if Img 1 is closer to the trained green value than Img 2
        Return Image 1 More Natural
    else
        Return Image 1 More Natural
    end
end
```

5.6 Sorting the Collection: The sorting function implements the worst case O(n*log n) comparison sorting algorithm, merge sort. Merge sort is a divide and conquer algorithm that takes a collection of values and divides it into sub-collections of size one, then proceeds to recursively sort and merge the divided sub-collections. Our implementation finds the middle of the collection, takes the ceiling of that value to ensure it is an integer, and splits the collection into sub-collections from 1: middle, and middle+1:end.



The sub-collections continue to be split until they have a size of one. The adjacent sub-collections, right and left, are then sorted and merged recursively until the whole collection is sorted and merged. We replaced the standard comparison with our relative comparison.

5.7 Downloading Images from Flickr: We wanted to get photos from online so we did not have a fixed set of images and knew that our sorting application would be capable of handling online collections. We used images from the photo sharing site Flickr that were similar to our selected quality of urban and natural. To interface with Flickr we used their public API [4]. We wrote the program for downloading images in Java since they didn't directly have an API for MatLab.

Our Java program is passed a command line argument which is the text for the search that is used to find source images. This can be entered by command line input in MatLab before running. The Java App then downloads the first one hundred images from that search into the source folder. They are mostly under 500 pixels in either dimension. We used a special image URL to grab low res versions of all the images. This solved the problem of downloading user copy protected images and made them consistent in size. Making sure the images were somewhat low resolution was also important in lowering runtime. We did also add code that resizes images if they are too big.

After the one hundred source images are downloaded it performs a hard-coded search for "(nature AND green) –portrait." These images are used in the color training function and help our algorithm learn an approximate green color scheme. Fifty training images are downloaded to the training folder.

5.8 Main Application: Our main script puts all these pieces together. First it prompts the user as to how many of the images he or she would like sorted. This is useful if you don't have the time to sort all 100 images. The value can be in (1,100). Regardless of how many images you would like to sort, the program downloads all

100 images. After that, it prompts for a search term for finding source images. Our default search term is `'(landscape, OR city, OR urban, OR natural) -portrait'` which yields search results of both natural and urban images and takes out most images with people in the shot. It then downloads fifty natural training images.

After the images are downloaded they are read into MatLab and the training images are used to train the algorithm for natural color schemes and then merge sort is called on the collection of images which performs the qualitative sorting using our relative comparison functions. After the images have been sorted from natural to urban, they are written to a separate sorted directory. The images are numbered 1 to 100 according to their sorted order.

6. Results

We have had quite successful results with our own qualitative image sorting algorithm. Given a set of one hundred images, only about ten or so were sorted incorrectly. There were only a couple of extremely misplaced images. It is difficult to quantify our results because people may differ in their comparison between very similar images. Based on our own comparisons and opinions on what qualities are natural and urban, our application has a success rate of around 90 percent.

To visually demonstrate our results we will show some sample comparisons as well as results from a small scale sort. The input and output of our sort of 100 images is in our project files. The input files are in ‘source’ and the output is in ‘sorted.’ The output images are named ‘Sort_n.jpg’ for n = 1 to 100, where ‘Sort_1.jpg’ is the most natural and ‘Sort_100.jpg’ is the most urban.

Figure 3: a. Original Image b. Vertical Edge Pixels c. Edge Pixels Above Threshold
This figure shows how we chose which points to use in the linear regression. Notice how this image is completely natural, since it consists of water, sky, rock and grass. It also has a very low number of pixels above the threshold. This is a nice preliminary result for finding or not finding lines in a natural image.

a. Original Image

b. Vertical Edge Pixels

c. Vertical Edge Pixels above threshold

Figure 4: Displayed are the original image, the image passed through a function to find the vertical edgepoints, and the final image c. Which only shows the pixels that are above the threshold of .3.These points will be used in the final linear regression to search for lines within the image. Notice that this urban example has many more possible line points than the natural example.

Input Image 1

Input Image 2

Input Image 3

Input Image 4

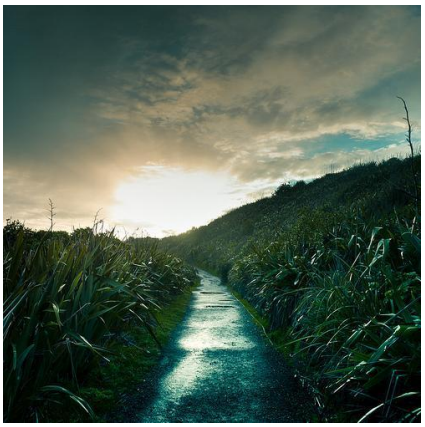
Input Image 5

Input Image 6

Figure 5: These are six natural, urban, and mixed images in an arbitrary order. They are most certainly already in natural to urban order. The six images are input by our application and sorted as described in section 5.

Output Image 1

Output Image 2



Output Image 3

Output Image 4

Output Image 5

Output Image 6

Figure 6: These are the same six images in the order determined by our image sorting application. No matter what order the images are to begin with, the output is always in this order. Given it is always subjective to define what makes an image more urban, we believe these six are in the same order most humans would sort them. The images are arranged from most natural to most urban. At this sample size our algorithm produces perfect results.

7. Conclusion

Relative attributes provide a way to describe images more completely than their binary counterparts. In our application we studied and implemented a sort for how urban or natural an image was when compared to another. Binary attributes could categorize the more extreme images, but binary attributes show weakness when trying to sort the images which contain elements of both. Relative attributes provide an avenue to qualitiatively assess an image, then relatively compare that image with other similar images allowing us to accurately sort collections by semantics. This project was successful, especially on the set of images downloaded from Flickr. Other types of natural images exist however, so improving on the color scheme analysis to include other natural color schemes such as snow, sunsets, and skies would be a possible extension. Line detection could also be improved as line detection via linear regression often finds lines that do not exist in the unaltered image. Qualitative comparisons via relative attributes are an excellent way to expand a computer's ability to describe similar scenes. They provide a pathway to sorting similar images in a way where binary attributes are not sufficient.

8. References

[1] D. Parikh, K. Grauman, Relative Attributes. (ICCV) 2011.

[2] Jensen, J. "Hough Transform for Straight Lines". Retrieved 15 December 2012.

[3] Hough Line Image. Retrieved 15 December 2012.

[4] Flickr API. Retrieved 28 November 2012.

Project Work:

All coding was done via in-person, pair programming.

Original Code: calcColor.m, colorTrain.m, compareImage.m, countLines.m, findedgepoints.m, main.m, GetFlickrImages.java

Adopted Code: mergeSort.m was partially adopted from [http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort #MATLAB], but was modified to use the compareImage comparison function.

Lines of Code Written: 500 Approx.

Debugging mostly by: Chris Lindner

All Code written by: Chris Lindner and Spencer Buyansky